Australian Government
Department of Defence
Defence Science and
Technology Organisation

Casbar User's Guide

Rowan J. Gollan, Brendan T. O'Flaherty, Peter A. Jacobs

Division of Mechanical Engineering
The University of Queensland

Ian A. Johnston

Weapons Systems Division
Defence Science and Technology Organisation

DSTO–GD–0594

ABSTRACT

The Collaborative Australian Ballistics Research code, `Casbar`, is a simulation tool for the analysis of the interior ballistics of guns. The code solves a two-phase, axisymmetric form of the governing equations for the flow of gas and particulates in the gun, and accommodates multiple projectiles within the simulation. `Casbar` is also suitable for investigating intermediate ballistics, and can alternatively be used as a general compressible flow solver. `Casbar` supports user-customised types of deterred or undeterred propellant grain, flexible definition of initial conditions and ignition sources, and various constitutive submodels for simulating interphase drag and intergranular stress. This document, the *Casbar User's Guide*, explains the use of the code and available options, and provides a worked example with corresponding input files.

Published by

Defence Science and Technology Organisation
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: (08) 8259 5555
Facsimile: (08) 8259 6567

AR No. 014-651
November, 2009

APPROVED FOR PUBLIC RELEASE

Contents

1 Introduction

`Casbar` is a suite of simulation tools that can be used to analyse the interior ballistic process in gun systems. The analysis is based on solving a governing set of conservation equations that describe the two-phase flow within a gun chamber. The equations are solved by discretising in a finite-volume manner. Thus `Casbar` is considered a computational fluid dynamics (CFD) tool.

This document is a manual on the use of `Casbar` — it provides information about input preparation, running simulations and post-processing. Additionally, some example cases are explained as tutorials. This manual only includes enough theory so that the input options are explained clearly. For more information on the theory behind the `Casbar` program, see Gollan and coworkers (2007)[1].

2 Overview of the simulation procedure

Setting up a simulation is mostly an exercise in writing a text-based description of your gun system. This specification file is a Python file with a `.py` extension. Also, the description of the propellant grain(s) appears in a separate Python file. By using a separate file for the propellant description, you can build up a library of propellant types and re-use the specifications without the errors of "copying-and-pasting" from one simulation file to another. Having prepared a problem specification file and propellant description file, the general steps for a simulation are as follows:

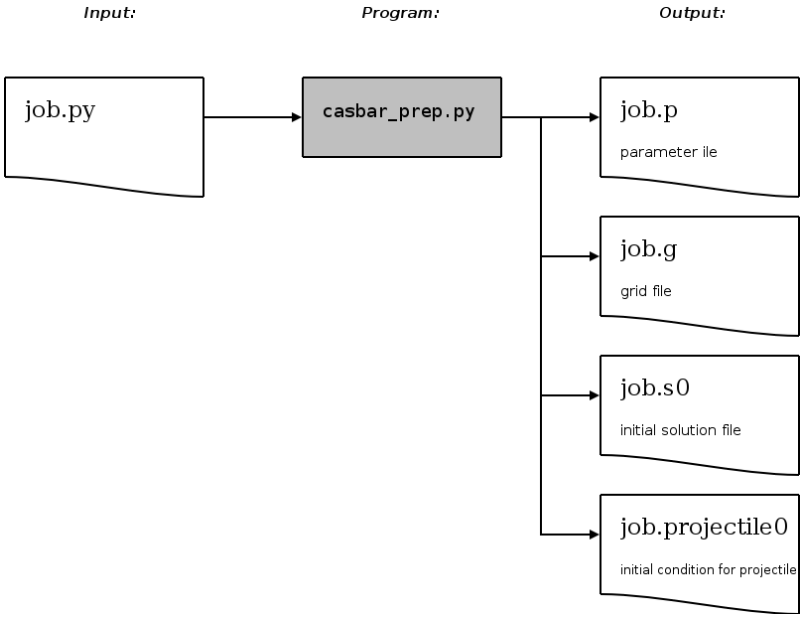
1. Prepare the propellant data file with the command
   `> prepare_propellant.py propellant.py propellant.dat`
   where `propellant.py` is the propellant description file prepared by hand and `propellant.dat` is the machine-generated output file in INI format that is used by the main simulation program. The instructions for preparing a propellant description file are detailed in Section 3.2.

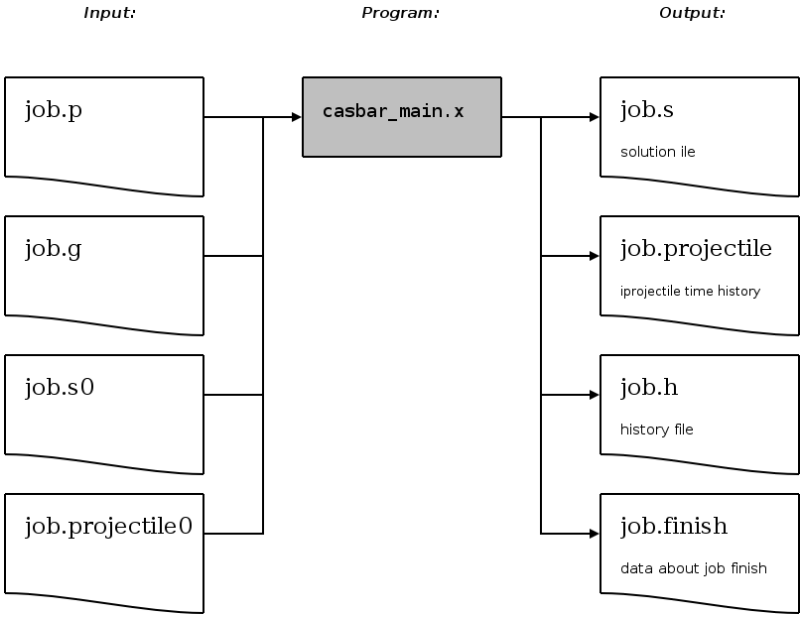
2. Prepare the main simulation files with the command
   `> casbar_prep.py -job=`*job*
   The italics word *job* should be replaced by the chosen name for your job. The command does not require that you type the `.py` extension. In this case it would look for a file named `job.py` in the working directory.

   The preparation program writes out various text files for use as input for the main simulation program:

   - `.p` file: This is the parameter file written in INI format. Normally, you would not need to create one of these parameter files manually. It is handy though to edit one or two parameters in the file without rerunning the simulation program. For example, you may wish to change the CFL number or edit how frequently the program writes out complete field solutions.

[1] Gollan, Johnston, O'Flaherty and Jacobs, *Development of Casbar: a Two-phase Flow Code for the Interior Ballistics Problem*, 16th Australasian Fluid Mechanics Conference (2007).


Input:
Program:
Output:
job.py
casbar_prep.py
job.p
parameter ile
job.g
grid file
job.s0
initial solution file
job.projectile0
initial condition for projectile


Input:
Program:
Output:
job.p
job.g
job.s0
job.projectile0
casbar_main.x
job.s
solution ile
job.projectile
iprojectile time history
job.h
history file
job.finish
data about job finish

- `.g` file: This file specifies the vertices which comprise the grid. Each of the blocks is listed sequentially in this file. Do not attempt to hand edit this file.
- `.s0` file: This file specifies the inital solution (flow field) for the simulation. It has a structured format which lists the conditions in every cell. Do not attempt to hand edit this file.
- `.`projectile0 file: The initial conditions for the projectile(s) if present are listed in this file.

Additionally some data files relating to the gas properties and possibly a look-up table for an igniter flux boundary condition may be created depending on what was requested in the problem specification.

3. The main simulation routine is run using the C++ program `casbar_main.x`.
   `> casbar_main.x -job=`*job*

4. The postprocessing step is somewhat specific based on what is desired. This step is documented in Section 4.

3 Constructing input files

As mentioned earlier, there are two Python files that the user needs to construct (with a text editor): (1) the problem specification file, and (2) the propellant(s) description file. All of the other input files are created based on the instructions in these two user input files. Commonly we refer to user input in these Python files as "Python-level" input. The actual simulation routine `casbar_main.x` parses INI-style files for input — these INI files are created by the preparation programs.

3.1 Problem specification file

The problem specification file is a Python file that is used to define the gun problem of interest. The input file is loaded by another Python program, `casbar_prep.py`. The controlling program, `casbar_prep.py`, begins by setting up some default global data (default timestep, default CFL, for example) and then executes the user's input file to get the specific parameters for the job. Thus a user's script can override the defaults provided by `casbar_prep.py`. In addition, `casbar_prep.py` is expecting that the user's script also provides details about the flow conditions, flow domain and other details of the simulation. In general, the order of declarations is unimportant in the user's script though there are some constraints:

- A gas model, grain burning model and intergranualar stress model must be set before a flow condition.
- A flow condition must be set before a block definition.

- Geometric entities are required to construct a block and so must be set before the block definitons.

With that in mind, a recommended order of problem specification is:

1. Simulation control parameters such as timestep, frequency of solution writing and maximum simulation time.
2. Gas model.
3. Grain burning model.
4. Intergranular stress model.
5. Interphase drag model.
6. Flow conditions.
7. Flow domain: geometry and block definition.
8. Projectile specification.
9. Igniter zone specification (optional).
10. History locations for data recording (optional).

In the next sections, each of these items is described in detail. It may be helpful to flick forward to page 30 to view an example input file in order to give some context to the following discussion.

3.1.1 Simulation control parameters

The simulation control parameters which affect global aspects of the simulation are stored in an object called `gdata`[2]. Upon entry to the user's script the `gdata` object is already initialised and certain defaults are set. The user can then override the defaults by setting the appropriate object attribute. The user sets an attribute by using a simple assignment statement. For example, to set the simulation title and initial timestep, the following two statements would be used:

```
gdata.title = "My gun simulation"
gdata.dt = 1.0e-5
```

A list of the most commonly used control parameters are given in Table 1. This table gives the name of the attribute, the type of value and the available options pertaining to that attribute. Each parameter listed in Table 1 is set the in the user's script with an assignment of the form:

```
gdata.param = value
```

The table also lists the default values for the various parameters where applicable. If a parameter is missing from the input script it will receive this default value.

[2]The `gdata` object is an instance of the `TwoPhaseGlobalData2D` class which is defined in `casbar_prep.py`.

Table 1: Description of simulation control parameters.

Parameter	*Type*	*Description*
`title`	string	The title string may be used to give a unique identifier to the simulation. This string is picked up in a number of places in the simulations routines.
`problem_type`	string	This is used to set which set of physical processes that `Casbar` will consider. The available options are: `"interior_ballistics"` **(default)** This problem type solves the complete interior ballistitics problem including processes such as gas and particulate transport, grain combustion and ignition modelling. `"gas_transport"` `Casbar` can actually be used as single-phase code for compressible flow problems by selecting this problem type. `"particulate_transport"` This problem type is used to test the transport of the particulate phase. `"two_phase_shock_problem"` This problem type is for verification puposes and does not solve a flow problem of practical interest for the user. `"closed_vessel"` This problem type simulates a closed vessel with no flow processes; only grain combustion occurs. This problem type is used during code testing and provides a convenient means to exercise the grain combustion module. `"drag_only"` When "drag only" is selected, the two-phase flow problem with drag interaction is computed. None of the other physical processes of the interior ballistics process are considered. `"piston_solver"` This solves a single-phase (gas flow) problem with piston motion included.

Parameter	*Type*	*Description*
`two_phase_system`	string	This option relates to the governing equations used to solve the problem. There is a subtle distinction between `problem_type` and `two_phase_system`. The `problem_type` parameter selects which physical processes are simulated. The `two_phase_system` selects the set of conserved variables. Presently there is only one option: `"Gough"` **(default)**, and as such it may be omitted. This parameter is present so that in future versions different sets of governing systems may be easily selected.
`axisymmetric_flag`	integer	There are two options: `1` **(default)** axisymmetric geometries, $y = 0$ is taken as the symmetry line. `0` for planar geometries.
`gas_flux_calc`	string	There are two flux calculators implemented for the gas phase transport problem: `"ausmdv"` **(default)** Recommended. `"ausm"`
`particulate_flux_calc`	string	There are two flux calculators implemented for the particulate phase transport problem which parallel the gas phase flux calculators: `"ausmdv-p"` **(default)** Recommended. `"ausm-p"`
`x_order`	integer	This parameter controls the order of accuracy used by the spatial reconstruction: `1` Low-order reconstruction. Cell-centred values are taken as interface values. `2` **(default)** Higher-order reconstruction. A piecewise parabolic segment is used to reconstruct interface values and a limiter is applied.

Parameter	*Type*	*Description*
`t_order`	integer	This parameter controls the order of time integration accuracy: `1` Euler method of update (first order). `2` **(default)** Predictor-corrector method (second order).
`cfl`	float	This value sets the Courant-Friedrichs-Lewy (CFL) number for the numerical methods. The default value is 0.5.
`dt`	float	This is the *initial* time step used. The default value is 1.0e-6 s. The time step will change during the simulation based on the CFL criterion. If the simulation fails very early, it might be helpful to reduce this initial timestep by an order of magnitude.
`dt_plot`	float	This parameter governs how frequently a complete flow field solution is recorded. It is a value in seconds in simulation time. Be careful not to select a value that is too frequent as it is possible to fill your hard disk by writing out too many snapshots of the flow field.
`dt_history`	float	This parameter controls how often the data in history cells and projectile state are written to file. This value is often smaller then `dt_plot` as it does not take much disk space to record information at a few selected cells.
`max_time`	float	This is the maximum simulated flow time that the simulation should run for.
`max_steps`	integer	This is the maximum number of steps that the simulation should take. This value is set in case the flow simulation runs into trouble and starts taking very small time steps.

3.1.2 Gas model

The gas model is selected by calling the `set_gas_model` method of the `gdata` object. The format for that call is:

```
gdata.set_gas_model(gas_name, gas_input_file)
```

where `gas_name` is a string specifying the type of gas model and `gas_input_file` is a string specifying a file name which contains the accompanying data for the gas model. There are numerous gas models available but listed here are those of most interest for the interior ballistics problem:

`"Noble_Abel_gas"` A single-species gas with real gas effects accounted for by a co-volume value.

`"Noble_Abel_gas_mix"` A mixture of the gases where each component is described as Nobel-Abel gas.

`"ideal_gas"` An ideal gas with calorifically perfect behaviour (single-species).

`"ideal_gas_mix"` A mixture of the aforementioned ideal gas.

The second aspect of the gas model specification is the `gas_input_file`. These files are INI-type files which contain the data about the specific gases. It is possible to prepare them by hand but it is often easiest to use one of the provided convenience functions which will create the input file directly. Here is a list of convenience functions — the detailed documentation for some of these functions follows.

- `create_Noble_Abel_gas()`
- `create_Noble_Abel_gas_mix()`
- `create_ideal_gas()`
- `create_ideal_gas_mix()`

`create_Noble_Abel_gas()`:
The `create_Noble_Abel_gas()` function is used to create an input file for a Nobel-Abel gas based on user-supplied gas parameters.

```
create_Noble_Abel_gas(R=287.0, gamma=1.4, name="Noble_Abel_gas", b = 0.001,
    Prandtl=0.72, Lewis=1.0, mu_ref=17.89e-6, T_ref=273.1, S=110.4, q=0.0,
    filename="Noble_Abel_gas.dat" )
```

The keyword arguments are (default values are in the function signature):

- `R`: specific gas constant in J/kg/K
- `gamma`: ratio of specific heats
- `name`: a label for the gas (of no real importance just for user's convenience)
- `b`: co-volume for gas in m^3
- `Prandtl`: the value for Prandtl number
- `Lewis`: the value for Lewis number
- `mu_ref`: a reference viscosity in Pa.s used in the Sutherland viscosity law
- `T_ref`: a reference temperature in K used in the Sutherland viscosity law
- `S`: the Sutherland constant in K

- `q`: a heat release value for the gas in J/kg
- `filename`: the name for the file to be created which will hold the gas data

The default values shown here (which are applied by calling `create_Noble_Abel_gas()` without any arguments) are for air with a co-volume of $0.001\,\mathrm{m}^3$ (which may or may not be appropriate for your case).

```
create_Noble_Abel_gas_mix():
create_Noble_Abel_gas_mix(file_list, filename="Noble_Abel_gas_mix.dat")
```

The Noble-Abel gas mix is comprised of Noble-Abel gas components. One would usually create the component input files by calling `create_Noble_Abel_gas()` for each component. The `create_Noble_Abel_gas_mix()` function then assembles an input file based on the component input files which are listed in `file_list`. The keyword argument `filename` is used to specify the name of the data file for the gas mixture.

An example should hopefully make this clear. Suppose you wish to specify a mixture of two gases with Noble-Abel behaviour. The first gas represents the propellant products and the second gas is the ambient air. You would proceed as follows:

1. Create the Noble-Abel gas `dat` file for the component gases: call `create_Noble_Abel_gas()` twice
2. Create the input data file for the mixture of two gases: call `create_Noble_Abel_gas_mix()` once
3. Declare the gas model and input file: call `gdata.set_gas_model()` once

Thus the snippet in your script file would be:

```
# 1. Create component gases
create_Noble_Abel_gas(name="propellant gas", R=390.3, gamma=1.27,
                      b=0.0010838, filename="propellant_gas.dat")
create_Noble_Abel_gas(name="Air", R=287.0, gamma=1.4, b=0.001,
                      filename="air.dat")
# 2. Create the data file for the gas mixture
create_Noble_Abel_gas_mix(["propellant_gas.dat", "air.dat"],
                          filename="Nobel_Abel_gas_mix.dat")
# 3. Declare the gas model and input file
gdata.set_gas_model("Noble_Abel_gas_mix","Noble_Abel_gas_mix.dat")
```

This example also highlights that keyword arguments can appear in any order, provided all required keywords are supplied. Thus you will note that the order in this example is different to the function signature from above. This is a feature of Python syntax — it is not peculiar to this input script or the `Casbar` program.

If you had already created `Noble_Abel_gas_mix.dat` in another simulation, you could just copy that file to your current working directory and proceed directly to step 3, selection of the gas model. Remember that steps 1 and 2 are just convenient ways to create the input file, `Noble_Abel_gas_mix.dat` — you could have copied or created this by any other means so long as it is a vaild input file.

Finally, the functions `create_ideal_gas()` and `create_ideal_gas_mix()` are used in the same way as their Noble-Abel counterparts except that the `create_ideal_gas()` does not accept a `b` parameter.

3.1.3 Grain burning model

The grain burning model describes the combustion properties of the various types of propellant grains. The specification of grains can become quite complex as the input allows for multiple grain types and multiple layering of solid types within grains. For this reason, the grain input file is prepared from a stand-alone script with the program `prepare_propellant.py`. This procedure is described fully in Section 3.2. In the simulation input script, the user only needs to specify the name of the grain input file. So assuming the grain input file has been previously created with `prepare_propellant.py`, the grain burning model is declared using:

```
gdata.set_grain_model(grain_file)
```

where `grain_file` is a string denoting the name of the grain input file.

3.1.4 Intergranular stress model

The intergranular stress model is set per grain type and the grains are numbered from $0 \ldots N-1$ where N is the number of grain types.[3] Thus a declaration using the `set_igs_model()` method of the `gdata` object should appear for each grain type.

```
gdata.set_igs_model(index, igs_model, igs_input_file)
```

where `index` is an integer identifying the grain type, `igs_model` is a string giving the intergranular stress model name and `igs_input_file` is a string giving the name of the input file for the specified stress model. The currently available intergranular stress models are:

- `"Gough_stress_model"`
- `"Koo_Kuo_model"`
- `"Kuo_Summerfield_model"`

[3] In common with Python and C/C++ conventions, numbering begins from zero. This is consistent throughout `Casbar` ; blocks, cells, species, and so on, are always numbered from zero.

Each of these models requires an accompanying input file to completely specify the model. Similar to the gas model input, there are certain convenience functions available to create the input files. They are:

- `create_Gough_stress_model_input()`
- `create_Koo_Kuo_stress_model_input()`
- `create_Kuo_Summerfield_stress_model_input()`

Thus the usual sequence of calls in the user script is to use one of these convenience functions to create an input file, and then declare the intergranular stress model with `gdata.set_igs_model()`.

`create_Gough_stress_model_input()`:
In the rheological model proposed by Gough for the intergranular stress, there are a number of parameters which are dependent on the grain. The equations for stress and associated granular wave speed are:

$$R = \rho_p a_1^2 \epsilon_0^2 \left(\frac{1}{\epsilon_g} - \frac{1}{\epsilon_0} \right) \tag{1}$$

and

$$a_p = \begin{cases} a_1(\epsilon_0/\epsilon_g) & \epsilon_g \leqslant \epsilon_0 \\ a_1 \exp[-\kappa(\epsilon - \epsilon_0)] & \epsilon_0 < \epsilon_g < \epsilon_* \\ 0 & \epsilon_g \geqslant \epsilon_* \end{cases} \tag{2}$$

where a_1, ϵ_0, κ and ϵ_* are empirical constants based on the properties of the granular bed.

The user may set each of these parameters by using the following function

```
create_Gough_stress_model_input(eps0, eps_star, a1, kappa,
                                const_wave_speed, filename)
```

where

- `eps0` is the settling porosity (often taken as the initial porosity), ϵ_0 *(float)*
- `eps_star` is the model parameter ϵ_* *(float)*
- `a1` is the model parameter a_1 in m/s *(float)*
- `kappa` is the model parameter κ *(float)*
- `const_wave_speed` is a Boolean (`True` or `False`) indicating whether a constant wave speed assumption should be used. If it is set true, the value given as `a1` is used as the granular wave speed, otherwise wave speed is computed using Equation 2.
- `filename` is a string for the data file into which the model parameters will be written.

`create_Koo_Kuo_stress_model_input()`:
The Koo and Kuo stress model calculates intergranular stress and wave speed from:

$$R = \begin{cases} -\rho_p C^2 \frac{\epsilon_g}{\epsilon_c} \left(\frac{\epsilon_c - \epsilon_g}{1-\epsilon_g} \right) & \epsilon_g \leqslant \epsilon_c \\ 0 & \epsilon_g > \epsilon_c \end{cases} \tag{3}$$

and

$$a_p = C_{\text{ref}} \frac{\epsilon_c}{\epsilon}. \tag{4}$$

The user is required to supply the model parameters C_{ref} and ϵ_c. This may be done by calling the following function:

```
create_Koo_Kuo_stress_model_input(C_ref, eps_c, filename)
```

Following the established pattern, `filename` is the name of the file into which the model parameters are written.

`create_Kuo_Summerfield_stress_model_input()`:
In the Kuo and Summerfield model, integranular stress is calculated as:

$$R = \begin{cases} \frac{\kappa \left[\frac{1}{1-\epsilon_c} - \frac{1}{1-\epsilon_g} \right]}{1-\epsilon_g} & \epsilon_g < \epsilon_c \\ 0 & \epsilon_g \geqslant \epsilon_c \end{cases} \tag{5}$$

The user needs to select the model parameters ϵ_c and κ. The wave speed calculation is the same as the Koo and Kuo model and as such the user specifies a value for C_{ref}. Thus an input file for the Kuo and Summerfield intergranular stress model is created using:

```
create_Kuo_Summerfield_stress_model_input(C_ref, eps_c, kappa, filename)
```

3.1.5 Interphase drag model

The interphase drag model is presently implemented as a global model to calculate the exchange of momentum between the gas phase and particulate phase due to drag. If there are multiple grain types present, the momentum is shared between various grain types based on their relative volumes in a given finite-volume cell. The interphase drag model is declared by calling the method `set_drag_model()` of the object `gdata`:

```
gdata.set_drag_model(drag_model, drag_input_file)
```

where `drag_model` is the name of a specific model for the interphase drag and `drag_input_file` is an input file for the model. Presently there are two options for interphase drag model:

- `"Ergun_drag_model"`

- `"zero_drag"` – not really a model but may be used to "turn off" interphase drag terms.

The Ergun drag model only requires a single parameter: a critical porosity, a value which allows the calculation to vary between modelling a packed bed or a fluidised bed. It may seem like overkill to create an input file just to specify one parameter. The justification is that future implementations may include more compilcated drag models which require more input parameters. So by using a file based input for this simple Ergun drag model the input will remain consistent when more complicated models become available. The function call to create the Ergun model input file is:

```
create_Ergun_drag_model_input(eps0, filename)
```

where `eps0` is the critical porosity mentioned earlier.

3.1.6 Flow conditions

A flow condition is a complete specification of the flow state at some point in time and space — thermodynamic state of the gas; gas phase velocity; stress state of the grains (loading density); and velocity of component grain types. A flow condition, built from a `FlowCondition` object, is often used to set initial conditions in the domain and boundary conditions at the edge of the domain such as a specified flux boundary condition.

First we describe the `ParticulateCondition` object which is used to specify the state of a single grain type. The `FlowCondition` object is composed of `ParticulateCondition` objects for each grain type as well as gas phase information.

A `ParticulateCondition` may be intialised as:

```
initial_loading = ParticulateCondition(index, u=0.0, v=0.0, ld=1000.0, r=0.0)
```

where

- `index:` is an integer specifying which grain this condition applies to
- `u:` is the x-velocity (axial) in m/s
- `v:` is the y-velocity (radial) in m/s
- `ld:` is the loading density of the grain type in kg/m^3
- `r:` is the regression distance of a single grain of the given grain type in m. Usually this value is 0.0 for unburnt grains, however, a positive value may be specified to simulate already partially burnt grains.[4]

Given that some `ParticulateCondition` objects have been instantiated, you can declare a flow condition using:

[4]This might be useful for patching the solution of one simulation into a larger domain.

```
initial = FlowCondition(p=None, T=None, rho=None, u=0.0, v=0.0, mf=[1.0,],
                        particulate_conditions=[None])
```

where

- `p`: is the gas pressure in Pa
- `T`: is the gas temperature in K
- `rho`: is the gas density in kg/m^3
- `u`: is x-velocity of the gas in m/s
- `v`: is y-velocity of the gas in m/s
- `mf`: is a list of component mass fractions. The values in this list should sum to 1.0.
- `particulate_conditions`: this is list of previously named `ParticulateCondition` objects. If a grain type is non-existent in a certain region (for example, intially ahead of the projectile), the Python keyword `None` may be given in the list. You must still list a condition for each grain type even if that condition is `None`. When the program receives `None` it will put zero mass of that grain type in the flow condition.

Note only two state variables for the gas should be specified: that is, choose only two out of pressure, temperature and density. If you specify all three, one of the values will be ignored and the thermodynamic state will be computed based on only two of the values. The code attempts to compute the state based on what values it finds and it tries, in order, to use (1) pressure and temperature; followed by (2) pressure and density; and finally (3) temperature and density.

3.1.7 Block definition of the flow domain

Most of the effort required to set up a simulation goes into defining the "body-fitted" grid of finite-volume cells that completely fills the flow domain. This grid is block structured, with each block defined by four edges (NORTH, EAST, SOUTH and WEST) fitted to the actual edges of the flow domain.

To define a block in your input script, create a `Block2D` object as:

```
my_block = Block2D(parametric_surface, nni, nnj,
                   cf_list, bc_list, fill_condition,
                   hcell_list, label)
```

where

- `parametric_surface`: is a region of 2D space bounded by four edges. See Section 3.1.7.3 for a guide to constructing a surface.

- `nni:` is the number of finite-volume cells in the i-direction. Note that, when placing one block against another, the blocks must conform in
  - the number of cells along corresponding edges
  - the clustering of those cells along the edges
  - the path defining the corresponding edges.
- `nnj:` is the number of finite-volume cells in the j-index direction.
- `cf_list:` which stands for cluster functions list is a list of `Function` objects that specify a (possibly) nonuniform distribution of cells along a particular edge of the `parametric_surface`. The order that the edges are listed in is NORTH, EAST, SOUTH, WEST. If this option is omitted, all edges receive a uniform distribution of cells.
- `bc_list:` is a list of boundary conditions that are applied to the edges in the order NORTH, EAST, SOUTH, WEST. If this option is omitted, all boundaries are treated as walls[5]. The available boundary conditions are described in Section 3.1.7.2.
- `fill_condition:` accepts either a `FlowCondition` object with which to define the initial flow state within the block volume or a user-defined function that varies in space to define the flow state. See Section 3.1.6 for defining a suitable `FlowCondition`. A discussion about user-defined fill functions follows this list.
- `hcell_list:` is a list of (i, j)-tuples specifying which cells should be monitored at simulation time. Data from the specified cells will be written to a "history" file for the simulation and may be used at the postprocessing stage to provide flow data as if there was a sensor located in the cell. As always, cell numbering begins from zero.
- `label:` is an optional text label for the block. This label will be embedded in the block definition and some of the postprocessing programs may use it.

If using multiple blocks, the block connections need to be specified. This is most easily achieved by calling the automated `identify_block_connections()` function after declaring the blocks.

3.1.7.1 User-defined fill functions A user may define a Python function that specifies how a block should be filled based on spatial variations. This can be used to initialise non-uniform flow fields (like propellant loading at one end only) or to transfer an old solution onto the new grid. The rules for the function are simple:

1. The function accepts two parameters, `x` and `y`, in that order which represent the x and y spatial positions (in physical coordinates) in the flow field.
2. The function returns an object of type `FlowCondition`.

[5]Certain boundaries may later be converted to connection boundaries if, after all the blocks have been specified, the `identify_block_connections()` function is called.

Note when returning the `FlowCondition` object it is useful to use the keyword argument `add_to_list=False`. This prevents the program from storing all of the temporary flow conditions created by the function call from being recorded in the global list of flow conditions.

An example of a user-defined fill function is given here. It simply initialises a propellant bed in the left-end of the domain, the chamber, below $x = 0.0$. In the right-end, the barrel, ambient air conditions are given. Note the function MUST accept `x` and `y` even if it only varies in one spatial dimension.

```
propellantloaded = ParticulateCondition(0, u=0.0, v=0.0, r=0.0, ld=913.47)
def fill_function(x, r):
    if x < 0.0:
        return FlowCondition(p=0.1e6, u=0.0, v=0.0, T=294.0,
                             mf=[0.0, 1.0, 0.0],
                             particulate_conditions=[propellantloaded],
                             add_to_list=False)
    else:
        return FlowCondition(p=0.1e6, u=0.0, v=0.0, T=294.0,
                             mf=[0.0, 1.0, 0.0],
                             particulate_conditions=[None],
                             add_to_list=False)
```

Alternatively, we could have named the two flow conditions earlier in the script and avoided needing to use the `add_to_list=False` argument. This is shown here.

```
propellantloaded = ParticulateCondition(0, u=0.0, v=0.0, r=0.0, ld=913.47)
propellantIC = FlowCondition(p=0.1e6, u=0.0, v=0.0, T=294.0,
                             mf=[0.0, 1.0, 0.0],
                             particulate_conditions=[propellantloaded])
barrelIC     = FlowCondition(p=0.1e6, u=0.0, v=0.0, T=294.0,
                             mf=[0.0, 1.0, 0.0],
                             particulate_conditions=[None])
def fill_function(x, r):
    if x <= 0.0:
        return propellantIC
    else:
        return barrelIC
```

These two examples give the equivalent initial flow field in the block.

3.1.7.2 Boundary conditions The boundary conditions for blocks may be set in the block definition as a list of conditions (`bc_list`) or they may be set after a block definition using:

```
my_block.set_BC(EAST, Extrapolate_boundary_condition())
```

In this method, the first argument specifies which boundary (NORTH, EAST, SOUTH or WEST) and the second argument is the boundary condition to apply.

The boundary conditions are all derived types of the abstract C++ class `Boundary_condition`. The constructors are made available at the Python-level input via SWIG. The available boundary conditions are:

- `Wall_boundary_condition()` **(default)** is a reflecting wall boundary condition.
- `Extrapolate_boundary_condition()` assumed supersonic outflow where the ghost-cell flow properties are simply copies of the adjacent interior cell properties.
- `Common_boundary_condition()` this is used to specify that an edge has an internal connection to another block. Normally the user doen't need to specify this as the `identify_block_connections()` will take care of applying `Common_boundary_conditions` in the right places.
- `Igniter_flux_boundary_condition(filename)` specifies a spatially and temporally varying flux boundary condition. The specified flux is intended to mimic the effect of igniter material discharge. The spatially and temporally varying nature of the flux boundary is handled through a look-up table given as the argument `filename`. This look-up table is most easily created using the `create_igniter_lut_bc_file()` convenience function which is documented in Section 3.1.9.2. The spatial variation along a boundary is only treated in one-dimenion. The following are the dimension of interest for each of the edges:
  - NORTH: x-dimension varies
  - EAST: y-dimension varies
  - SOUTH: x-dimension varies
  - WEST: y-dimension varies

  For example, when treating a SOUTH boundary condition with an igniter flux, the x position of the cell-centres along the SOUTH boundary are used to "look-up" the appropriate flux at that point.

3.1.7.3 Constructing surfaces: geometry The top-level geometry description given to the grid generator is in terms of "parametric surfaces". These are regions of 2D space that may be traversed by a set of parametric coordinates $0 \leq r < 1$ and $0 \leq s < 1$. These surfaces can be constructed as a "boundary representation" from lower-dimensional geometric entities: paths and points.

The most fundamental class of geometric object is the `Vector` (or `Vector3` as it is defined in the C++ module `libgeom2`). A `Vector` represents a point in 3D space and has the usual behaviour of a geometric vector (as opposed to the `vector` collection class in C++). If you want a point to be rendered with a label, you can define it as a `Node`. Examples of use include: a = `Vector`(x, y) and b = `Node`(x, y, `label='B'`).

The next level of dimensionality is the `Path` class. A path object is a parametric curve along which points can be specified via the single parameter $0 \leq t < 1$. Types of paths that are available include:

- `Line`(a,b): a straight line between points a and b.
- `Arc`(a,b,c): a circular arc from a to b around centre, c.
- `Arc3`(a,b,c): a circular arc from a through b to c. All three points lie on the arc.
- `Bezier`($[b_0, b_1, ..., b_n]$): a Bezier curve from b_0 to b_n.
- `Polyline`($[p_0, p_1, ..., p_n]$): a composite path made up of the segments p_0, through p_n. The individual segments are reparameterised, based on arc length, so that the composite curve parameter is $0 \leq t < 1$.
- `Polyline2`(.. arbitrary list of `Vectors` and `Paths` ..): a composite path made by joining points and paths with straight lines in the sequence listed. *Note:* The user's script will need to import this special object if needed. Before using, add the line: `from cfpylib.geom.path import Polyline2`
- `Spline`($[b_0, b_1, ..., b_n]$): a cubic spline from $b0$ through $b1$, to bn. A `Spline` is actually a specialised `Polyline`.

The user may construct a `ParametricSurface` which uses transfinite interpolation from four paths which represent the NORTH, EAST, SOUTH and WEST boundaries of a surface. The function to construct this is `make_patch` and it accepts four path objects in the order of NORTH, EAST, SOUTH and WEST. The ends of paths should coincide at the approriate corners otherwise the grid generator will complain. This function returns a `ParametricSurface` suitable for the the `Block2D` object to construct a grid. The function call is:

```
make_patch(north, east, south, west)
```

3.1.8 Projectile specification

Projectiles may be specified using the `Projectile` object. You can initialise a projectile by calling the initialiser with the following options:

```
Projectile(m, D, xL0, xR0,
           v0=0.0, rifling_twist=0.0, rog=0.0,
           bore_resistance_x=[0.0,],
           bore_resistance_p=[0.0,],
           constant_velocity=False,
           positive_velocity=False,
           vanish_at_x=VERY_LARGE_X,
           name="")
```

Note that the initialisation of a `Projectile` requires four mandatory arguments: `m`, `D`, `xL0` and `xR0`. The rest of the arguments are keyword arguments — if not specified the defaults are applied as shown. The parameters for the `Projectile` object are:

- `m:` is the mass of the projectile in kg
- `D:` is the diameter of the projectile in m
- `xL0:` is the starting position of the projectile (WEST face of projectile) in the x-direction (axial) in m
- `xR0:` is the fininishing position position of the projectile (EAST face of the projectile) in the x-direction (axial) in m
- `v0:` is an initial x-velocity of the projectile in m/s
- `rifling_twist:` is the number of turns per calibre. If set to zero, a smooth bore is simulated.
- `rog:` is the radius of gyration in m
- `bore_resistance_x:` is a list of x-ordinates (in m) which specify break points for the interpolation of bore resistance as a function of axial position. If the list only has one value then there is nothing to use for interpolation and so a constant value of bore resistance is applied everywhere.
- `bore_resistance_p:` is a list used in conjunction with `bore_resistance_x` list to specify the variation of bore resistance as a function of axial distance. This list contains the value of resistance in pressure, Pa, at the locations corresponding to the `bore_resistance_x` list. The number of entries in `bore_resistance_x` and `bore_resistance_p` must match or an error will be raised.
- `constant_velocity:` is a Boolean which will set the projectile's motion at constant x-velocity, `v0`, if set to true. When set to false **(default)**, the projectile moves under the influence of the pressure forces acting on its faces.
- `positive_velocity:` is a Boolean value (`True` ot `False`). If true, the projectile will only be allowed to have positive velocities. If a negative velocity is computed based on flow conditions, the projectile velocity will be set to zero. If set to false **(default)**, the projectile update proceeds as normal.
- `vanish_at_x:` is an x-ordinate in m which specifies at position at which the projectile is removed (or "vanishes") from the simulation. Its intent is to allow for the removal of the projectile at some point in the far field.
- `name:` is an optional name for the projectile.

3.1.9 Ignition modelling

There are two ways to model the ignition process in the code: (1) as a volume of influence, and (2) as flux at a boundary. The two methods are not mutually exclusive in a given simulation.

3.1.9.1 Ignition zone An `IgnitionZone` object may be used to specify a region in the flow where some mass and energy are added to the gas in order to mimic the effect of ignition. The declaration of an `IgnitionZone` has the following signature:

```
IgnitionZone(point0, point1,
             rdot, chem_energy, mf,
             t_start, t_end, label="")
```

where

- `point0:` is a `Vector3` object which locates the bottom left corner of the ignition zone.
- `point1:` is a `Vector3` object which locates the upper right corner of the ignition zone.
- `rdot:` is the rate of mass addition per unit volume of physical space, $\dot{\rho}$, in $kg/m^3/s$. Note this is per total volume available, not only that available to the gas.
- `chem_energy:` is the chemical energy the injected gas is created with in J/kg.
- `mf:` is a list of mass fractions which identifies which gaseous species the injection of mass goes into. The values in this list should sum to 1.0.
- `t_start:` is the starting time for the ignition zone to take effect in s.
- `t_end:` is the finishing time for the ignition zone's influence in s. After this simulation time is exceeded the ignition zone no longer has any effect.
- `label:` an optional label.

You may specify multiple igniton zones. The implementation is quite naïve about the interacting ignition zones. The criteria for applying the effects of ignition are simply this:

1. Does the cell-centre of a finite-volume cell lie within the bounding box (`point0`, `point1`)? and
2. Is the current simulation time between `t_start` and `t_end`?

If these two criteria are satisfied, then the cell with have mass and enegy added at the rate dictated by `rdot` and `chem_energy`. If a given cell satisfies this criteria for more than one `IgnitionZone` then the effect will be accumulative.

If you wanted to mimic the effect of varying ignition rates in a given region, you could declare multiple `IgnitionZone` objects that acted over different times by using different values for `t_start` and `t_end` in each of the declarations.

3.1.9.2 Igniter flux at a boundary We saw earlier that specifiying an igniter flux boundary condition involved setting the boundary condition to `Igniter_flux_boundary_condition(filename)` where `filename` is a look-up table describing how the flux varies in space and time. Now we discuss how to construct a look-up table through the use of some supplied convenience functions.

A look-up table file for the igniter flux boundary condition is created using:

```
create_igniter_lut_bc_file(flux_function, s_locations,
                           t_locations, filename)
```

where

- `flux_function`: is a user-defined Python function that accepts `s` and `t` which are spatial and temporal values respectively, and returns the fraction exposed `fe` and a `FlowCondition` object. This function is exlained in more detail below, but essentially describes how the flux at the boundary varies spatially and temporally.
- `s_locations`: is a list of spatial locations which will be used when constructing the look-up table. For a SOUTH boundary this would be a list of x-ordinate values, for an EAST boundary this would be a list of y-ordinate values, and so on. The user chooses how fine or coarse the look-up table is by the number and distribution of values in the list. The values should sequentially increase.
- `t_locations`: is list of time values which will be used when constructing the look-up table. Similarly to `s_locations`, the user chooses the granularity of the look-up table interpolation by choosing the distribution of `t_locations`.
- `filename`: is the name of the file in which the look-up table will be created. This file is later handed to the boundary condition during specification. It is usual to name this file with a `.gz` extension because this function creates a gzipped textfile.[6]

The user-defined function has some minimal stipulations:

1. It must accept a spatial and temporal variable in that order: `def f(s, t)`
2. It must return a tuple which contains the fraction of area exposed at that point and the flow condition: `return (fe, FlowCondition)`.

The flux is calculated based on the area through which the `FlowCondition` is applied and the actual condition itself. The `FlowCondition` is specified in the global frame of reference, so a `v`-velocity for gas flow will move in the radial direction. Also, the interface area is set by the boundary along which the flux condition is supplied. If you set the boundary condition on a $y = 0.0$ boundary in an axisymmetric simulation you will not get any flux at all because the interface area on the $y = 0.0$ line is zero.

We now look at an example to see it all in action. In this example, the igniter flux is modelled along a length of 25 cm beginning from $x = -1.0$ m. The flux begins at $t = 0.0$ and finishes when $t = 2.5$ ms. In that region the available surface area for material flux is only 50%. The code in our script would be as follows.

[6]The **zlib** library is used to create the file.

```
# ------------------------------------------------------
# 1. Define the function which represents the flux
# ------------------------------------------------------
def flux_function(x, t):
    # First test within time constraint
    if t > 2.5e-3:
        # Flow condition is arbitrarily at atm conditions
        # because the fraction exposed is 0.0
        return (0.0, FlowCondition(p=1.0e5, T=300.0,
                                   u=0.0, v=0.0, mf=[1.0],
                                   particulate_conditions=[None],
                                   add_to_list=False))
   # Next test if outside x range
   if (x < -1.0) or (x > -0.75):
        # Again return an arbitrary flow condition with the
        # fraction exposed equal to 0.0
        return (0.0, FlowCondition(p=1.0e5, T=300.0,
                                   u=0.0, v=0.0, mf=[1.0],
                                   particulate_conditions=[None],
                                   add_to_list=False))
   # So, therefore, we are in the 25cm of igniter region
   # and in the time period of interest
   return (0.5, FlowCondition(p=1.0e6, T=1000.0,
                              u=0.0, v=100.0, mf=[1.0],
                              particulate_conditions=[None],
                              add_to_list=False)

# ------------------------------------------------------
# 2. Specify the points in the look-up table in x and t
# ------------------------------------------------------
x_locations = [ -1.0 + i*2.5e-3 for i in range(11) ]
t_locations = [ 0.0 + i*2.5e-4 for i in range(12) ]

# ------------------------------------------------------
# 3. Construct the look-up table
# ------------------------------------------------------
create_igniter_lut_bc_file(flux_function, x_locations,
                           t_locations, "lut_bc.dat.gz")

# ------------------------------------------------------
# 4. Use on the SOUTH boundary of my_block
# ------------------------------------------------------
my_block.setBC(SOUTH, Igniter_flux_boundary_condition("lut_bc.dat.gz"))
```

The internal implementation uses bi-linear interpolation (between space and time) to compute the appropriate flux based on cell-centre positions in the boundary condition calculation. The user should take care at the edges of their look-up table: constant

extrapolation is used at the edges of the table, ie. the closest edge value is taken as the value. The ramifications are that in this example we ensured there was a time interpolant point in the flux equals zero regime. If this had not been the case, the last point may have left the flux "turned on" for all time after $t = 2.5\,\text{ms}$.

We'll repeat the warning in another way. Just because the user-defined function turns fluxes on and off in the appropriate way at the appropriate times does not mean that the internal effect is guaranteed. The selection of spatial and temporal locations for the interpolation points also influences the behaviour. The easiest way to avoid any surprises is to place interpolation points close to, but either side of any intended boundaries in your flux function.

3.1.10 Specifying history locations

When constructing a `Block2D`, you may optionally specifiy a `hcell_list` which allows you nominate specific cells at which the history of flow data should be recorded. It is often more convenient so specify (x, y) coordinates rather than the (i, j) index values which are grid specific. The `HistoryLocation` object allows you to use physical coordinates and may be declared using:

```
HistoryLocation(x, y, label="")
```

where `x` is the x ordinate, `y` is the y ordinate and `label` is an optional label. The label is used to help you identify the cell in the `casbar_history_cells.list` file. If you declare one or more `HistoryLocation` objects, a file, `casbar_history_cells.list`, is created listing the information about the located cell: block number and (i, j) indices.

The searching algorithm will locate the nearest cell-centre to the chosen (x, y) values. The searching algorithm has no knowledge about the extents of the actual flow domain. Therefore, it is possible to requrest a location beyond the edge of the domain — the returned value will simply be the closest cell to that location. The history file indicates the actual location of the history cell in the columns `x_found` and `y_found`.

3.1.11 Summary: simulation checklist

In this section we again review the list presented in Section 3.1, which detailed a recommended sequence of declarations in the input file. However, we now present it as a checklist and indicate the appropriate objects to initialise and functions to call.

- ☐ Declare simulation control parameters such as flux calculators and initial timestep. Each declaration has the form: `gdata.param = value`.
- ☐ Select the gas model. After creating an appropriate input file, declare the gas model by calling the method: `gdata.set_gas_model(model_name, input_file)`.
- ☐ Specify the grain burning model. Given that a grain input file has been prepared (see Section 3.2), use the method: `gdata.set_grain_model(input_file)`.

- ☐ Select the intergranuluar stress model (for each grain type) and the set the appropriate model parameters. First create an input file using one of the convenience functions, then use the member method:
  `gdata.set_igs_model(index, model_name, input_file)`.
- ☐ Set the interphase drag model. An input file may be created using a convenience function and then the model is declared by calling the method: `gdata.set_drag_model(model_name, input_file)`.
- ☐ Set flow conditions using the `FlowCondition` construct. This is only necessary if you are using flow conditions which fill entire regions. If you elect to use a block fill function, you might defer specification of those flow conditions to that function.
- ☐ Specify geometry and build blocks. Declare `Nodes`. Construct `Paths` built from those `Nodes`. Create surface patches based on the `Paths`. Finally, construct blocks (`Block2D` objects) which define the flow domain.
- ☐ Optionally, declare a number of `Projectile` objects.
- ☐ Optionally, declare a number of `IgnitionZone` objects.
- ☐ Optionally, declare a number of `HistoryLocation` objects.

3.2 Propellant grain description file

The propellant grain description file `propellant.py` is a Python file that is used to define the propellants of interest. The propellant grain description file is processed by another Python program, `prepare_propellant.py`, which subsequently generates an INI format data file suitable for loading by `Casbar` itself. As noted in Section 2, the following command line processes the `propellant.py` input file into the machine-generated output file `propellant.dat` which can be read by `Casbar` :

```
> prepare_propellant.py propellant.py propellant.dat
```

In practice, the user may choose to automate this step by including it within the main Python `job.py` job file:

```
os.system("prepare_propellant.py propellant.py propellant.dat")
```

Irrespective of how `propellant.dat` is generated, an instruction needs to be included in `job.py` to tell `Casbar` to load it:

```
gdata.set_grain_model("propellant.dat")
```

In this example we have used the filenames `propellant.py` and `propellant.dat`, however the user is free to choose any legal filename which might better suit their needs.

Definition of the propellants in the `propellant.py` propellant grain description file is achieved in two parts:

- First, a set of solids are defined to represent each distinct energetic material formulation in the simulation.
- Second, each propellant grain type is defined in terms of its geometry and its composition of one or more layers. Each layer is composed of one—or a mixture—of the declared solid types.

3.2.1 Definition of the energetic material solid types

Each distinct energetic material is defined in the manner of the following example:

```
example_solid = Solid("my_example_solid_propellant_material",
                      density=1578.0,
                      flame_temperature=2585.0,
                      combustion_energy=3.7369e6,
                      gas_massf=[1.0, 0.0, 0.0],
                      burn_rate_min_p=[0.0, 200.0e6,],
                      burn_rate_param_a=[0.00078385,0.001,],
                      burn_rate_param_b=[0.0, 0.0,],
                      burn_rate_param_n=[0.9, 1.0,])
```

The initial arguments are

- the name of the solid.
- `density`: True density of the solid in kg/m^3.
- `flame_temperature`: Flame temperature of the solid in K.
- `combustion_energy`: The intensive internal energy e of the solid's combustion products, in J/kg.
- `gas_massf`: An array of mass fractions, defining the gaseous products produced by combustion of the solid. The species order reflects the order of definition described in Section 3.1.2, and the sum of the mass fractions should equal unity.

The subsequent arguments define the linear burn rate of the solid material. The linear burn rate r (in m/s) is defined by Vielle's law, $r = aP^n + b$. Multiple sets of coefficients and exponents, corresponding to different pressure ranges, may be used to obtain a higher fidelity burn rate model if desired.

- `burn_rate_min_p`: An array of minimum pressures for which each set of burn rate parameters is valid, in Pa.
- `burn_rate_param_a`: An array of Vielle's law coefficients for each pressure range, in MPa^{-n} m/s.
- `burn_rate_param_n`: An array of Vielle's law exponents for each pressure range.

- `burn_rate_param_b`: An array of Vielle's law parameters for each pressure range, in m/s.

The user can alternatively define blocks of constant burn rate by specifying $a = 0$ and defining b as desired for each pressure range. It is important to note that, unlike all other `Casbar` inputs, the units of a above are not base SI. This is to reflect that most published burn rate coefficients for Vielle's law correspond to P in MPa.

Where a simulation is to incorporate a deterred propellant solid, the user should define an additional unique solid with burn rate properties modified to match that of the deterred material.

Once all solids are defined, they are declared using:

```
declare_solids([example_solid])
```

3.2.2 Definition of the propellant grain types

The actual propellant grain types are defined in terms of their geometry, the solid energetic materials they contain, and the grain ignition temperature. Initially, the propellant geometry and ignition temperature is defined in the manner of the following example:

```
example_grain = Grain("my_example_propellant_grain",
                      geom_type="GRAIN7PERFCYL",
                      outer_diameter=11.43e-3,
                      perforation_diameter=1.143e-3,
                      length=25.4e-3,
                      ignition_temperature=444.0)
```

The `ignition_temperature` is expressed in K, and represents the local gas temperature that would cause the grain to ignite. `Casbar` supports a number of grain geometries. The currently available `geom_type` keywords, and the required input dimensions for each, are now described.

- `GRAINCYLINDER`: A solid cylinder or cord. Specify `outer_diameter` of the cord and cord `length`.
- `GRAIN1PERFCYL`: A single-perforated cylindrical grain. Specify `outer_diameter` of the cylinder, `perforation_diameter` and `length`.
- `GRAIN7PERFCYL`: A seven-perforated cylindrical grain. Specify `outer_diameter` of the cylinder, `perforation_diameter` and `length`. Webs are assumed to be of equal size.
- `GRAIN19PERFCYL`: A nineteen-perforated cylindrical grain. Specify `outer_diameter` of the cylinder, `perforation_diameter` and `length`. Webs are assumed to be of equal size.

- `GRAINSPHERE`: A solid ball. Specify `outer_diameter` of the sphere.

Each grain type may contain one or more solid energetic materials. The solids (or mixtures of solids) are arranged in layers, where each layer is defined by its depth from a free surface where combustion occurs. In the case of perforated grains, the perforation surfaces are also treated as free surfaces. The following example shows the definition of a grain comprised wholly of a single solid, and thus containing a single layer:

```
example_grain.add_layer(solid_massf=[1.0],
                        layer_start=0.0)
```

The array `solid_massf` denotes the mass fractions of solid materials in that layer, in the order defined in Section 3.2.1. The keyword `layer_start` defines the start of the layer, expressed as depth from the initially unburnt free-surfaces of the grain. Multiple layers with varying mass fractions can be defined, for example, to approximate impregnation of one solid material through another.

Finally, the propellant grains must be declared to `Casbar` using

```
declare_grains([example_grain])
```

4 Postprocessing tools

4.1 Extracting field data: `casbar_post.py`

The postprocessing program `casbar_post.py` may be used at the command line to extract field data from the simulation domain. The command-line options are explained here.

```
> casbar_post.py --job=JOBNAME --format=FORMAT --output=OUTPUT
                 [--time=TIME|--initial|--final|--all]
```

`-job=`*JOBNAME*
: *JOBNAME* is the base file name that is used for your simulation. The program will look for `.s` and `.p` files based on this name.

`-format=`*FORMAT*
: *FORMAT* is one of: `save`, `vtk` or `tecplot`.

`-output=`*OUTPUT*
: *OUTPUT* is the base part of the output file name. The program will add the appropriate extension based on the format and time option.

One only of the following options must be specified:

`-time=`*TIME*
: *TIME* is the time in seconds at which the field is desired. The program will select the first field solution that is *greater* than the specified time value.

`-initial`
: The initial flow field is extracted and written to a file *OUTPUT*`-initial` plus appropriate extension. The `.s0` file is used as the data.

`-final`
: The solution file (`.s`) is scanned for the final solution and this is written to *OUTPUT*`-final` plus appropriate extension.

`-all`
: All available field solutions are written out in sequence. This may be useful for creating animations.

4.2 Extracting history data: `casbar_history.x`

The history cell data is recorded in the `.h` file for all history cells in the flow field. The `casbar_history.x` program may be used to extract the data for a specific history cell and write the data in a form suitable for plotting. It is important that the user is aware how many history cells are in the simulation because the history extraction program needs this value in order to correctly pull out the data.

```
> casbar_history.x --parameter-file JOBNAME.p --input JOBNAME.h
                   --output OUTPUT  --ncell <1>  --cell <0>
```

`-parameter-file` *JOBNAME.p*
: This option indicates the appropriate parameter file.

`-input` *JOBNAME.h*
: This option is used to specify the history file.

`-output` *OUTPUT*
: This is the name of the file, chosen by the user, into which the history data for the selected cell will be written.

`-ncell` *ncells*
: *ncells* is the number of history cells which appear in the file *JOBNAME.h* The default value is 1.

`-cell` *cell_no*
: *cell_no* is the number of the specific history cell for which the data is required. The numbering of cells is from $0 \ldots ncells - 1$. The default value is 0.

4.3 Extracting profile data: `casbar_prof.py`

The user may extract a line of data from the flow field using the `casbar_prof.py` tool. The line follows a constant *i* or *j* index through the grid and so does not necessarily correspond to line of constant *x* or *y* value. The resulting output file is in a form ready for plotting. The data in each of the columns is identified by the fields in the first line of the output file.

```
casbar_prof.py --job=JOBNAME --output=OUTPUT
               [--i-line=<i_index>|--j-line=<j_index>]
               [--block-list=<BLK_LIST>]
               [--time=TIME|--initial|--final|--all]
```

`-job=`*JOBNAME*
: *JOBNMAE* is the base file name that is used for your simulation. The program will look for `.s` and `.p` files based on this name.

`-output=`*OUTPUT*
: *OUTPUT* is the base part of the output file name. The program will append the extension `.prof` to this name.

The user must select either an `-i-line` or a `-j-line`:

`-i-line=`*i_index*
: *i_index* is the integer value of constant *i*-index along which the profile is extracted. This would usually be used to select a vertical line throughout the grid.

`-j-line=`*j_index*
: *j_index* is the integer value of constant *j*-index along which the profile is extracted. This would usually be used to select a horizontal line throughout the grid.

Additionally, the user must select one of the following time options:

`-time=`*TIME*
: *TIME* is the time in seconds at which the profile is desired. The program will select the first solution that is *greater* than the specified time value.

`-initial`
: The profile is extracted from the initial flow field and written to a file *OUTPUT*`-initial.prof`. The `.s0` file is used as the data.

`-final`
: The solution file (`.s`) is scanned for the final solution and the extracted profile is written to *OUTPUT*`-final.prof`.

`-all`
: All available field solutions are processed and the appropriate profile is written out to files in sequence. Useful for creating animations.

4.4 Separating the data for multiple projectiles

The `Casbar` program stores the information for *all* projectiles in the `.projectile` file. Each line begins with an index indicating which projectile that line of data applies to. In the case of a single projectile, it is easy to use the `.projectile` file directly for plotting. When you have multiple projectiles you may wish to separate the data into separate files. A trivial *awk*[7] program as shown below may be used from the command line:

```
awk -v proj=1 '$1 == proj { for (i=2; i<=NF; i++) \
   printf "%s ", $i; printf "\n"; }' \
   < jobname.projectile > output
```

In this example, the data for the second projectile (index = 1, therefore `proj=1`) is extracted from the input file `jobname.projectile` and the data is written to a file named `output`.

5 Example: The AGARD gun

5.1 AGARD gun description

The "AGARD gun" is a synthetic test case, which has previously been used for performing code-to-code comparisons in several TTCP efforts, including KTA 4-13 and KTA 4-38. See, for example, Woodley, *Modelling the ignition of 40mm gun charges*, 22nd International Symposium on Ballistics, Vancouver, 2005.

The gun chamber diameter and bore diameter are constant at 132 mm, and the bore resistance is a constant 13.79 MPa. The projectile base is initially located 762 mm downstream from the breech. In this example, the igniter is assumed to vent uniformly throughout the full chamber diameter, in the region between the breech and 127mm downstream of the breech. Heat loss to the barrel is neglected. The propellant consists of cylindrical 7-perforated grains. Thermal properties of the propellant and other relevant data are prescribed and shown at Table 2.

5.2 Listing of `agard_propellant.py`

The following listing shows the `Casbar` propellant description file used to define the AGARD gun propellant properties and geometry. Note that we specify that the propellant produces only one product gas, with the corresponding properties of that gas described in the job file. While in reality the propellant combustion would produce multiple species, for simplicity we simply use an homogenous product gas with properties matching that of the mixture of those species.

[7] The `awk` programming language is available on most linux distributions.

Table 2: *AGARD Gun Data (Woodley, 2005)*

Gun calibre (mm)	132 (constant)
Initial position of projectile from breech face (mm)	762
Travel of projectile (mm)	4318
Distance from breech face to muzzle (mm)	5080
Bore resistance (MPa)	13.79 (constant)
Projectile mass (kg), flat base	45.359
Propellant mass (kg)	9.5255
Propellant solid density (g/cc)	1.578
Propellant geometry	cylindrical 7-hole
Propellant grain length (mm)	25.4
Propellant grain diameter (mm)	11.43
Propellant perforation diameter (mm)	1.143
Propellant burn rate coefficient (cm/s/MPa^n)	0.078385
Propellant burn rate pressure index (n)	0.9
Propellant adiabatic flame temperature (K)	2585
Propellant ignition temperature (K)	444
Propellant thermal conductivity (W/s/K)	0.2218
Propellant thermal diffusivity (mm2/s)	0.08677
Propellant emissivity (-)	0
Propellant chemical energy (MJ/kg)	3.7369
Propellant molecular weight (g/mol)	21.3
Propellant specific heat ratio (-)	1.27
Propellant impetus (MJ/kg)	1.009
Propellant co-volume (cc/g)	1.0838
Propellant intergranular wave speed (m/s)	254
Igniter mass (kg)	0.2268
Igniter density (g/cc)	1.799
Igniter chemical energy (MJ/kg)	1.5702
Igniter molecular weight (g/mol)	36.13
Igniter specific heat ratio (-)	1.25
Igniter impetus (MJ/kg)	0.3926
Igniter adiabatic flame temperature (K)	1706
Initial temperature in chamber (K)	294
Initial pressure	atmospheric
Molecular weight of ambient air (g/mol)	29
Specific heat ratio of ambient air (-)	1.4

```
# A python description file for
# agard propellant.

agard_solid_propellant = Solid("agard_solid_propellant",
                               density=1578.0,
                               flame_temperature=2585.0,
                               combustion_energy=3.7369e6,
                               gas_massf=[1.0, 0.0, 0.0],  # [prop gas, air, primer]
                               burn_rate_min_p=[0.0,],
                               burn_rate_param_a=[0.00078385,],
                               burn_rate_param_b=[0.0,],
                               burn_rate_param_n=[0.9,])

declare_solids([agard_solid_propellant])

agard_propellant_grain = Grain("agard_propellant",
                               geom_type="GRAIN7PERFCYL",
                               outer_diameter=11.43e-3,
                               perforation_diameter=1.143e-3,
                               length=25.4e-3,
                               ignition_temperature=444.0)

agard_propellant_grain.add_layer(solid_massf=[1.0],
                                 layer_start=0.0)

declare_grains([agard_propellant_grain])
```

5.3 Listing of `agard.py`

The following listing shows the `Casbar` propellant job file used to define the AGARD gun simulation. The listing contains explanatory commenting throughout, preceded by the Python commenting `#` symbol. In addition, note that:

- The `import os` command is required in order to effect the processing of the propellant description file from within this Python job file.
- Various convenience variables (like `Diameter`) and functions can be defined to suit the user.
- The barrel is made longer than specified in the case definition, by the length of the projectile. This allows room for the projectile to fully exit the "real" muzzle location before the simulation stops. Otherwise, the simulation would end when the projectile nose reaches the end of the barrel.
- A medium resolution of 129 cell vertices in the x-direction, and 8 in the radial direction, is used.
- A Python function, `fill_function`, is used to provide the intial conditions for the entire solution domain. It uses each cell's x-location to determine whether it is to be filled by air (upstream of the projectile) or propellant (between breech and projectile base).

- The origin for coordinates has been chosen to correspond to the centre of the projectile base. This x-origin is arbitrary, however, and any other convenient point along the symmetry axis could have been used.

```
# AGARD idealized gun test case
#

import os

Diameter = 132.0e-3
r = Diameter/2.0
p_length = 0.381

job_title = "AGARD"
gdata.title = job_title
gdata.stringent_cfl = 1
gdata.two_phase_system = "Gough"
gdata.problem_type = "interior_ballistics"

#
# Drag model
#
settling_porosity = 0.42112
create_Ergun_drag_model_input(settling_porosity, "Ergun_drag_model.dat")
gdata.set_drag_model("Ergun_drag_model", "Ergun_drag_model.dat")

#
# Gas model
# 3 gases: propellant gas, air, primer
#
create_Noble_Abel_gas(name="agard propellant gas", R=390.3, gamma=1.27, b=0.0010838,
                      filename="agard_propellant_gas.dat")
create_Noble_Abel_gas(name="Air", R=287.0, gamma=1.4, b=0.001,
                      filename="air.dat")
create_Noble_Abel_gas(name="agard primer gas", R=230.1, gamma=1.25, b=0.001,
                      filename="agard_primer_gas.dat")
create_Noble_Abel_gas_mix(["agard_propellant_gas.dat", "air.dat", "agard_primer_gas.dat"],
                          filename="agard_gas_mix.dat")

gdata.set_gas_model("Noble_Abel_gas_mix","agard_gas_mix.dat")


#
# Propellant Grain
#
os.system("prepare_propellant.py agard_propellant.py agard_grain.dat")
gdata.set_grain_model("agard_grain.dat")

#
# Stress model
#
eps_star = 0.55 # dummy value as we're using constant wave speed
kappa = 1.0     # dummy value as we're using constant wave speed
a1 = 254.0      # m/s as specified in AGARD case
const_wave_speed = True
create_Gough_stress_model_input(settling_porosity, eps_star, a1, kappa, const_wave_speed,
                                "Gough_stress_model.dat")
```

```
gdata.set_igs_model(0, "Gough_stress_model", "Gough_stress_model.dat")

#
# Flow conditions
#
propellantloaded = ParticulateCondition(0, u=0.0, v=0.0, r=0.0, ld=913.47)

propellantIC = FlowCondition(p=0.1e6, u=0.0, v=0.0, T=294.0,  mf=[0.0, 1.0, 0.0],
                             particulate_conditions=[propellantloaded])

barrelIC     = FlowCondition(p=0.1e6, u=0.0, v=0.0, T=294.0, mf=[0.0, 1.0, 0.0],
                             particulate_conditions=[None])

#
# Block Geometry
#
a = Node(-0.762,            0.000, label="a")
b = Node( 4.318 + p_length, 0.000, label="b")
c = Node(-0.762,            r,     label="c")
d = Node( 4.318 + p_length, r,     label="d")

# Breech    Projectile                     Muzzle
#           Base (At Origin)
#   c         pppppp                        d
#   a        Opppppp                        b
#   < 762mm ><      4318mm + p_length       >

ab = Line(a, b)
cd = Line(c, d)
ac = Line(a, c)
bd = Line(b, d)

nx = 129
ny = 8

def fill_function(x, r):
    if x <= 0.0:
        return propellantIC
    else:
        return barrelIC

blk_0 = Block2D(make_patch(cd, bd, ab, ac),
                nni=nx, nnj=ny,
                fill_condition=fill_function,
                hcell_list=[(0,0)],
                label="blk_0")

blk_0.set_BC(EAST, Extrapolate_boundary_condition())

proj = Projectile(m=45.359, D=Diameter,
                  xL0=0.0, xR0=p_length,
                  v0=0.0, bore_resistance_p=[13.79e6],
                  bore_resistance_x=[0.0],
                  name="proj")

ignition_source = IgnitionZone(Vector(a.x, a.y), Vector(a.x+127.0e-3, a.y+r),
                               13049.73, 1.5702e6, [0.0, 0.0, 1.0],
                               0.0, 10.0e-3)
```

```
HistoryLocation(a.x+10.0e-3,  r)
HistoryLocation(a.x+750.0e-3, r)

#
# Job parameters
#
gdata.axisymmetric_flag = 1
gdata.gas_flux_calc = "ausmdv"
gdata.particulate_flux_calc = "ausmdv-p"
gdata.max_time = 20.0e-3
gdata.max_step = 300000
gdata.x_order = 2
gdata.t_order = 2
gdata.cfl = 0.25
gdata.dt = 1.0e-6
gdata.fixed_time_step = 0
gdata.print_count = 20
gdata.dt_plot = 2.0e-4
gdata.dt_history = 1.0e-4
```

5.4 Running the simulation

The following listing shows the operating system shell commands required to prepare and run the simulation, and extract history data from the results.

```
> casbar_prep.py --job=agard

> casbar_main.x --job=agard

> casbar_history.x -p agard.p -i agard.h -o history-breechmid.data --ncell 3 --cell 0
> casbar_history.x -p agard.p -i agard.h -o history-wall10mm.data --ncell 3 --cell 1
> casbar_history.x -p agard.p -i agard.h -o history-wall750mm.data --ncell 3 --cell 2
```

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA				1. CAVEAT/PRIVACY MARKING
2. TITLE Casbar User's Guide			3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)	
4. AUTHORS Rowan J. Gollan, Brendan T. O'Flaherty, Peter A. Jacobs and Ian A. Johnston			5. CORPORATE AUTHOR Defence Science and Technology Organisation PO Box 1500 Edinburgh, South Australia 5111, Australia	
6a. DSTO NUMBER DSTO–GD–0594	6b. AR NUMBER 014-651		6c. TYPE OF REPORT General Document	7. DOCUMENT DATE November, 2009
8. FILE NUMBER 2009/1101318/1	9. TASK NUMBER LRR 07/249	10. SPONSOR DSTO	11. No OF PAGES 36	12. No OF REFS 2
13. URL OF ELECTRONIC VERSION http://www.dsto.defence.gov.au/corporate/reports/DSTO–GD–0594.pdf			14. RELEASE AUTHORITY Chief, Weapons Systems Division	

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved For Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DSTO RESEARCH LIBRARY THESAURUS

Guns
Analysis
Internal Ballistics
Simulation Tools

19. ABSTRACT

The Collaborative Australian Ballistics Research code, `Casbar`, is a simulation tool for the analysis of the interior ballistics of guns. The code solves a two-phase, axisymmetric form of the governing equations for the flow of gas and particulates in the gun, and accommodates multiple projectiles within the simulation. `Casbar` is also suitable for investigating intermediate ballistics, and can alternatively be used as a general compressible flow solver. `Casbar` supports user-customised types of deterred or undeterred propellant grain, flexible definition of initial conditions and ignition sources, and various constitutive submodels for simulating interphase drag and intergranular stress. This document, the *Casbar User's Guide*, explains the use of the code and available options, and provides a worked example with corresponding input files.